Assertion-based debugging of imperative programs by abstract interpretation

François Bourdoncle

DIGITAL Paris Research Laboratory
85, avenue Victor Hugo
92500 Rueil-Malmaison — France
Tel: +33 (1) 47 14 28 22

Centre de Mathématiques Appliquées
Ecole des Mines de Paris, BP 207
06560 Sophia-Antipolis Cedex
France

bourdoncle@prl.dec.com

Abstract. Abstract interpretation is a formal method that enables the static determination (i.e. at compile-time) of the dynamic properties (i.e. at run-time) of programs. So far, this method has mainly been used to build sophisticated, optimizing compilers. In this paper, we show how abstract interpretation techniques can be used to perform, prior to their execution, a *static* and *automatic* debugging of imperative programs. This novel approach, which we call *abstract debugging*, lets programmers use assertions to express *invariance properties* as well as *inevitable properties* of programs, such as termination. We show how such assertions can be used to find the origin of bugs, rather than their occurrences, and determine *necessary conditions of program correctness*, that is, necessary conditions for programs to be bug-free and correct with respect to the programmer's assertions. We also show that assertions can be used to restrict the control-flow of a program and examine its behavior along specific execution paths and find necessary conditions for the program to reach a particular point in a given state. Finally, we present the *Syntox* system that enables the abstract debugging of *Pascal* programs by the determination of the range of scalar variables, and discuss implementation, algorithmic and complexity issues.

1 Introduction

Since most, if not all, programmers are unable to write bug-free programs, debugging has always been an important part of software engineering. The most common approach for debugging a program is to run this program on a well chosen set of examples and check that each run is bug-free, that is, that the program "behaves as expected". However, this approach has several severe shortcomings.

For instance, even with an extensive and carefully chosen set of examples, the method offers absolutely no guaranty that every part of the program's code has been tested under all possible conditions, which is unacceptable for mission-critical systems. Moreover, it is sometimes very difficult, with "post-mortem" debuggers such as *adb* or *dbx*, to find the *origin* of a bug just by looking at the current memory state right after the bug has occured.

Methods have been proposed to help programmers at this stage by allowing the reverse execution of programs, but these methods require that every assignment encountered during the execution of the program be memorized, which makes them only applicable to functional programs with few side-effects [21]. It would thus be desirable to have a framework allowing the *static*, *formal* and *automatic* debugging of programs. Even though this might seem impossible at first glance, since the general problem of finding all the bugs in a program is undecidable, we introduce in this paper an assertion-based framework that enables the static and automatic discovery of certain categories of bugs.

This framework is based on abstract interpretation, which is a formal method, pioneered by Patrick and Radhia Cousot [7, 10, 12], that enables the static and automatic determination of *safe* and *approximate* run-time properties of programs. So far, abstract interpretation has mainly been used in compilers to optimize, vectorize, determine the lifetime of dynamically allocated data structures (compile-time garbage collection), etc. The emphasis has thus been put on the efficient determination of general properties of correct executions of programs rather than on the determination of correctness conditions, and abstract interpretation has always been considered as a batch, optimization-oriented method, that should not be made accessible to the programmer.

In this paper, we propose a method where the programmer is allowed to insert assertions in the source-code of the program being debugged, and where violations of these assertions are treated as run-time errors. Two kinds of assertions can be used. *Invariant assertions* are properties which must *always hold* at a given control point, and are similar to the classical `assert` statement in *C* programs. *Intermittent assertions* are properties which must *eventually hold* at a given control point. Differently stated, intermittent assertions are *inevitable properties* of programs, that is, properties such that every execution of the program inevitably leads to the specified control point with a memory state satisfying the intermittent property.

For instance, the *invariant* assertion *false* can be used to specify that a particular control point *should not* be reached, whereas the intermittent assertion *true* can be used to specify that a particular control point *should* be reached. In particular, the termination of a program can be specified by inserting the intermittent assertion *true* at the end of the program. Invariant and intermittent assertions can be freely mixed, which gives the programmer a great flexibility to express correctness conditions of a program and test its behavior along certain execution paths.

This paper is organized as follows. In section 2, we give several examples of the categories of bugs that can be automatically discovered by abstract debugging and describe how a programmer can interact with an abstract debugger to locate these bugs. Then, in section 3 we give an intuitive presentation of the basic ideas and techniques of abstract debugging, and explain why and how they can be used to debug programs. Finally, in section 4, we present the prototype *Syntox* system that enables the abstract debugging of *Pascal* programs without procedure parameters by the determination of the range of scalar variables. We discuss implementation, algorithmic and complexity issues, show that even very simple properties such as the range of variables enable the determination of non-trivial bugs, and show that this system can be used to safely suppress most array bound checks during the execution of *Pascal* programs.

2 Examples

In this section, we exemplify the concept of abstract debugging on a few erroneous programs. A very common programming error consists in using out-of-bounds array indices in loops. For instance, the "For" program of figure 1 will obviously exit on a run-time error when accessing T[0], unless $n < 0$ at point ①. Moreover, if the index i ranges from 1 to n instead of 0 to n, then the program will abort when accessing T[101] unless $n \leq 100$ at point ①. Similarly, program "While" will loop unless b is $false$ at point ①, and program "Fact" will loop unless $x \geq 0$ at point ①.

It might seem quite difficult to discover these bugs automatically. However, an abstract debugger such a *Syntox*, described in section 4, will automatically discover and report the above necessary conditions of correctness. A compiler could use these conditions to issue a warning or generate a call to a specific error handler to do some clean-up and exit safely, or else could enter a special debugging mode to do a step-by-step execution of the program until the bug actually occurs.

The interesting fact about abstract debugging is that it predicts bugs *before* they actually happen, which permits a safe handling of these bugs. Further more, an abstract debugger always attempts to find the *origin* of bugs, rather than their occurrences, and *back-propagates* necessary conditions of correctness as far as possible in order to minimize the amount of information delivered to the programmer. Consequently, this feature makes abstract debugging much more useful than traditional methods and global flow analyzers such as *Lint* for instance, which is well known for the large number of warnings it generates.

For example, it is much more interesting to know that variable n of program "For" must be lower than 100 at point ① than to know that i must be less than 100 at point ② since the former test can be done once and for all after n has been read, whereas the latter must be done for every access to T. Moreover, if $n > 100$ at point ①, then it is *certain* that the program will either loop or exit on a run-time error in the future.

As we shall see in the next section, abstract debugging is a combination of *forward* and *backward* analyses of programs. Forward analyses mimic the forward (i.e. regular) execution of programs, whereas backward analyses mimic the *reverse* execution of programs and are responsible for the "discovery" and the "factorization" of the correctness conditions of programs.

As an example, consider program "BackwardAnalysis" of figure 1. Starting from the beginning of the program, a forward analysis will find 1) that $j \leq 3$ at point ③ and that unless $j \geq -1$, a run-time error will occur when reading T$[j+2]$, and 2) that $j \geq 4$ at point ④ and that unless $j \leq 100$, a run-time error will occur when reading T$[j]$. A forward analysis thus determines potential occurrences of bugs in a program. However, a backward analysis of the same program will successively show that in order for the program not to abort on a run-time error, the following properties must hold: $j \in [4, 100]$ at point ④, $j \in [-1, 3]$ at point ③, $j \in [-1, 100]$ at point ②, and finally $i \in [-1, 49]$ at point ①.

This last information can then be combined with the forward data flow, which shows that the post-condition $i \in \mathbb{Z}$ of the "read" procedure call does not imply the pre-condition $i \in [-1, 49]$ determined by the backward analysis. Hence, a warning can be issued to inform the programmer that if $i \notin [-1, 49]$ at point ①, then his program

```
program While;
var  i : integer;
     b : boolean;
begin
     i := 0; read(b);  ①
     while b and (i ≤ 100) do
    ②    i := i − 1
end.
```

```
program For;
var  i, n : integer;
     T : array [1..100] of integer;
begin
     read(n);  ①
     for i := 0 to n do
    ②    read(T[i])
end.
```

```
program Intermittent;
var  i : integer;
begin
     read(i); ①
     while (i ≤ 100) do
    ②    i := i + 1 ③
④
end.
```

```
program Select;
     var  n, s : integer;
     function Select(n : integer) : integer;
     begin
          if  (n > 10)  then
               Select := Select(n + 1)
          else if  (n > −10)  then
               Select := Select(n − 1)
          else if  (n = −10)  then
         ③    Select := 1
          else
               Select := −1
          end;
     begin
          read(n); ①
          s := Select(n);
          writeln(s); ②
     end.
```

```
program Shuttle;
     label 1;
     const N=100;
     var  i, j, tmp : integer;
          T : array [1..N] of integer;
begin
     for i := 1 to N − 1 do
     begin
          for j := i downto 1 do
               if  (T[j] > T[j+1])  then begin
                    tmp := T[j];
                    T[j] := T[j + 1];
                    T[j + 1] := tmp
               end else
                    goto 1;
     1:
     end
end.
```

```
program Fact;
     var x, y : integer;
     function F(n : integer) : integer;
     begin
          if  (n = 0)  then
               F := 1
          else  F := n * F(n − 1)
     end;
begin
     read(x); ①  y := F(x)  ②
end.
```

```
     program BackwardAnalysis;
          var  i, j : integer;
               T : array [1..100] of integer;
     begin
          read(i, b); ①  j := 2 * i + 1;
     ②    if  (j <= 3)  then
          ③    read(T[j + 2])
          else
          ④    read(T[j])
     ⑤
     end.
```

Figure 1: Examples

will *certainly* fail later on. Hence the *origin* of the bugs, i.e. the fact that the program does not test variable i after reading it, has been found, and a necessary condition of correctness, which is also sufficient in this particular case, has thus been discovered. Also, note that a further forward analysis would show that $j \in [4, 99]$ holds at point ④ for any correct execution of this program, which refines the property $j \in [4, 100]$ determined by the backward analysis.

As stated in the introduction, an important feature of abstract debugging is that programmers can freely insert *invariant assertions* and *intermittent assertions* in their programs to either statically check that important invariance properties are satisfied or check under which conditions a program eventually reaches a control point while satisfying a given property.

Intermittent assertions allow for a very powerful form of debugging. As an example, if the intermittent assertion $i = 10$ is inserted at point ② of program "Intermittent" of figure 1, then *Syntox* shows that a necessary condition for the program to *eventually* reach control point ② with $i = 10$ is that $i \leq 10$ at point ①.

The way intermittent assertions are handled *bottom-up* is easy to understand. In this particular case, an abstract debugger would start from the set $\{\langle ②, 10\rangle\}$ representing the program state $i = 10$ at point ②, and compute all the possible ancestors, namely $\langle ①, 10\rangle$, $\langle ③, 9\rangle$, $\langle ①, 9\rangle$, $\langle ③, 8\rangle$, $\langle ①, 8\rangle \ldots$, adding them one by one to the set of "correct states".

It is thus possible to determine the set of program states (and, in particular, of input states) from which a program eventually reaches a given control point, by simply inserting the intermittent assertion *true*, representing all the possible states, at this point. So for instance, if the intermittent assertion *true* is inserted at point ② of program "Select" of figure 1, *Syntox* shows that a necessary condition for the program to terminate is that $n \leq 10$ at point ①. Differently stated, if $n > 10$, then the program will certainly loop or exit on a run-time error.

Further more, if the invariant assertion *false* is inserted at point ③, *Syntox* shows that $n < -10$ at point ① is a necessary condition for the program to terminate without control ever reaching point ③. And finally, if the intermittent assertion $s = 1$ is inserted at point ②, *Syntox* shows that a necessary condition for this assertion to eventually hold is that $n \in [-10, 10]$ at point ①.

Invariant assertions can therefore be used to restrict the control flow and examine the behavior of a program along specific execution paths, and contrary to intermittent assertions, invariant assertions are handled *top-down*, that is, states which violate the invariants assertions, as well as all their ancestors, are removed, rather than added, from the set of correct states.

3 Abstract debugging

As stated in the introduction, abstract interpretation aims at computing safe approximations of *flow-insensitive* run-time properties of programs, that is, properties which hold at a given control point independently of the path followed to reach it. These properties, which are not limited to boolean properties, can be for instance the range or congruence

properties [15] of integer variables, or relational properties such as linear inequalities of the form $i \leq 2 * j + 1$ between the variables of a program.

The method is based on a characterization of program properties as *fixed points* of monotonic functions over complete lattices. For instance, if τ is a *predicate transformer* describing the operational semantics of a program, then for every program property ψ, $\tau(\psi)$ is a property characterizing the set of states reached after one program step executed from states satisfying ψ, and the *program invariant*, which characterizes the set of descendants of a set of input states satisfying a property ψ_0 is known to be the *least fixed point*, with respect to implication, of the function:

$$\psi \longmapsto \psi_0 \vee \tau(\psi)$$

or, equivalently, the least solution of the equation:

$$\psi = \psi_0 \vee \tau(\psi)$$

This equation simply states that program states reached during executions of the program started from an input state satisfying ψ_0 are either input states satisfying ψ_0 or descendants of other reachable states.

The previous equation reflects the forward execution of the program. However, forward execution is not sufficient for the purpose of abstract debugging. To see why, let us consider an *invariant assertion* Π_a and an *intermittent assertion* Π_e that one wishes to prove about the program. Two properties are of interest:

- The property $always(\Pi_a)$ which characterizes the set of states whose descendants satisfy Π_a.
- The property $eventually(\Pi_e)$ which characterizes the set of states for which there exists at least one descendant satisfying Π_e.

For instance, if there are input states which do not satisfy $always(\Pi_a)$ or, worse, if no input state satisfy $always(\Pi_a)$, then it is sure that the program is not correct with respect to Π_a, that is, every execution starting from an input state which does not satisfy $always(\Pi_a)$ will certainly lead to a state which does not satisfy Π_a.

Similarly, if there are input states which do not satisfy $eventually(\Pi_e)$, then every execution starting from an input state which does not satisfy $eventually(\Pi_e)$ will never reach a state satisfying Π_e.

So for instance, let Π_{out} and Π_{err} respectively characterize the sets of output states and the set of error states, and let $\overline{\Pi}$ denote the negation of property Π. Then $eventually(\Pi_{out})$ characterizes the set of states for which the program terminates, $eventually(\Pi_{err})$ characterizes the set of states leading to a run-time error, $always(\overline{\Pi_{out}})$ characterizes the set of states which either cause the program to loop or to exit on a run-time error, and $always(\overline{\Pi_{err}})$ characterizes the set of states which do not lead to a run-time error.

It can be shown [12] that if the program is deterministic, as it is the case for imperative languages, then $always(\Pi_a)$ is the *greatest* solution (w.r.t. implication) of the equation:

$$\psi = \Pi_a \wedge \tau^{-1}(\psi)$$

and $eventually(\Pi_e)$ is the *least* solution of the equation:

$$\psi \;=\; \Pi_e \;\vee\; \tau^{-1}(\psi)$$

where τ^{-1} is the predicate transformer describing the *backward semantics* of the program, that is, if ψ is a program property, then $\tau^{-1}(\psi)$ is the property which characterizes the set of direct *ancestors* of the states satisfying ψ. Note that, for technical reasons, we make the assumption that output states are fixed points of the transition relation defining the operational semantics of the program. We can see that the two properties of interest are defined in terms of the *backward* semantics of the program. The abstract debugging of a program can then be performed as follows.

a) Compute the *program invariant* **I** which represents the set of program states that can be reached during program executions which are correct with respect to the programmer's specifications Π_a and Π_e (see below).

b) Signal to the programmer every *input* state which does not satisfy **I** and every state satisfying **I** whose direct descendant does *not* satisfy **I**.

Step *b* thus determines a minimum set of "frontier" states that certainly lead to an incorrect execution of the program. These states are produced by the forward data flow but their descendants are not part of the backward flow. As illustrated in section 2, frontier states typically correspond to "read" statements or to entry points of loops. Hence, step *b* determines, as expected, the *origin* of bugs, rather than their occurrences.

As shown with program "BackwardAnalysis" in section 2, the program invariant **I** can be computed as the *limit* of the decreasing chain $(I_k)_{k\geq 0}$ (w.r.t. implication) defined by $I_0 = true$ and iteratively computed by applying the following steps in sequence:

1) Compute the characterization I_{k+1} of the set of descendants of input states satisfying I_k as the *least* solution of the forward equation:

$$\psi \;=\; I_k \;\wedge\; \big(\psi_0 \;\vee\; \tau(\psi)\big)$$

2) Compute the characterization I_{k+2} of the set of states satisfying I_{k+1} whose descendants satisfy Π_a as the *greatest* solution of the backward equation:

$$\psi \;=\; I_{k+1} \;\wedge\; \big(\Pi_a \;\wedge\; \tau^{-1}(\psi)\big)$$

3) Compute the characterization I_{k+3} of the set of states satisfying I_{k+2} for which there exists at least one descendant satisfying Π_e as the *least* solution of the backward equation:

$$\psi \;=\; I_{k+2} \;\wedge\; \big(\Pi_e \;\vee\; \tau^{-1}(\psi)\big)$$

Note that if the programmer only specifies one of the two assertions Π_a or Π_e, then only one of the two steps 2 or 3 has to be applied. Also, note that in practice, it is often sufficient to apply steps 1-2-3-1, although, in general, the chain $(I_k)_{k\geq 0}$ can be infinitely

strictly decreasing, i.e. $I_0 \Leftarrow I_1 \Leftarrow I_2 \cdots$, since each step can refine the previous one. For instance, if a backward propagation "removes" several erroneous input states, then the next forward propagation will eliminate the descendants of these input states, etc.

For example, if the *intermittent* assertion $y \leq 0$ is inserted at point ② of program "Fact", to determine if it is possible for the program to terminate with $y \leq 0$, *Syntox* shows after steps 1-3-1 that a necessary condition for this property to hold is that $x \geq 1$ at point ①. But if step 3 is applied once more, then *Syntox* shows that no correct program execution can satisfy this property. Therefore, it is proven that the *invariant* assertion $y \geq 1$ holds at point ②, that is, if control ever reaches point ②, then $y \geq 1$.

So far, we have assumed that programs are deterministic, but most programs are not deterministic. For instance, programs with "read" statements or logic programs are not deterministic, and program states can have several descendants. However, it can be shown that the above method remains valid, that is, that the condition stating that frontier states are incorrect is still a *necessary* condition of correctness, but is not sufficient in general.

For instance, if the statement "i := i + 1" of program "Intermittent" is replaced by "read(i)" then the conditions $i \leq 200$ at points ① and ③ and the condition $i = 200$ at point ④ are necessary conditions for property $i = 200$ to *eventually* hold at the end of the program, but these conditions are not sufficient, since the program might loop forever if the values read for variable i are always less than 100.

Of course, the method we have described is interesting from a mathematical point of view, but is not directly implementable, since fixed points over infinite domains are not computable in general. This is why abstract interpretation defines standard methods [3, 4, 5, 6, 7, 10, 11, 12] for *finitely* and *iteratively* computing *safe approximations* of **I**. These approximate invariants $I^{\#}$ describe true properties about the run-time behavior of the program, that is $I \Rightarrow I^{\#}$, but are not necessarily optimal.

Note that if the approximation of the program invariant **I** is necessary for the approach to be tractable, this approximation implies that the correctness conditions determined by an abstract debugger are *necessary* but not always sufficient, even for deterministic programs. For example, if a necessary and sufficient condition of correctness is that an integer variable x be such that $x \in \{1, 3, 5\}$, and the interval lattice is used to represent approximate properties, then $x \in [1, 5]$ is only a necessary condition for the program to be correct.

4 The Syntox system

The *Syntox* system is a prototype interprocedural abstract debugger that implements the ideas of section 3 for a subset of *Pascal*. This debugger can be used to find bugs that are related to the range of scalar variables, such as array indexing, range sub-types, etc. The interval lattice used to represent program properties is thus non-relational, but we have shown [4] that any relational lattice can be chosen, and that the results can be arbitrarily precise, even in the presence of aliasing, local procedures passed as parameters, non-local gotos and exceptions (which do not exist in *Pascal*).

Even though the interval lattice is quite simple, we shall see in section 4.5 that it allows to determine non-trivial bugs and program properties.

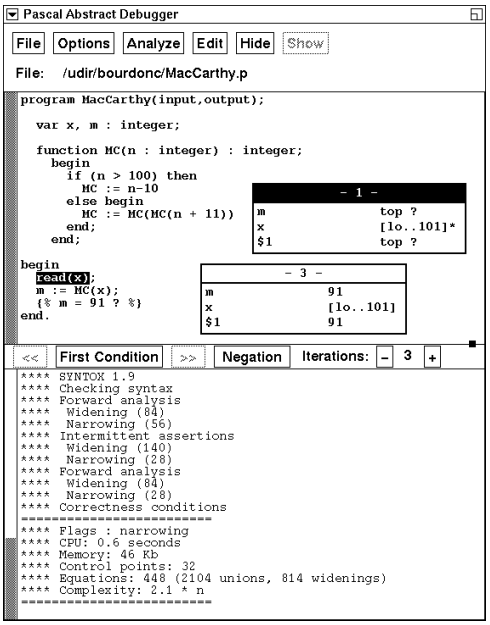

Figure 2: The *Syntox* system

4.1 Semantic issues

A problem that has to be solved to allow the abstract interpretation of *Pascal*-like languages is the *aliasing* induced by the formal reference parameters of procedures and functions, which create different variables with the same actual address. In order to increase the precision and reduce the complexity of the analysis, *Syntox* uses a non-standard, copy-in/copy-out semantics of first-order *Pascal* programs (i.e. programs without procedures passed as parameters) with jumps to local and non-local labels. This semantics, which is described in Bourdoncle [2, 4], determines the exact aliasing of programs, and is very well suited to abstract interpretation. We have shown that it is equivalent to the standard, stack-based semantics of *Pascal* [1]. We have also designed a version of this semantics for higher-order imperative languages that can be found in Bourdoncle [4], and shown that it is also equivalent to the standard semantics of the following classes of higher-order imperative programs with jumps to local and non-local labels:

- Second-order programs, i.e. programs where procedures which are passed as parameters to other procedures have non-procedural parameters only. Every *Wirth-Pascal* program [22] is second-order, but *ISO-Pascal* programs [18] can be higher-order.

- Higher-order programs with exceptions but without local procedures.

These classes can be shown to be sufficiently general to allow the abstract debugging of *C* programs with `setjmp` and `longjmp` statements (not considering the pointer-induced aliasing problems). As a matter of fact, exception handlers can be emulated by local procedure passed as parameter, and a `longjmp` statement is no more than a call to the exception handler which restores the `setjmp` context and branches to a local label of its enclosing procedure.

4.2 Language restrictions

Although the above theoretical results show that this could be done without any major problem, *Syntox* does not yet allow procedures to be passed as parameters to other procedures. Variant records and the "with" construct are not allowed in programs. Only the most standard *Pascal* library functions are predefined. Programs with pointers to heap-allocated objects are accepted, but are not always handled safely with respect to aliasing; other works on the abstract interpretation of heap-allocated data structures such as Deutsch [13, 14] could be used to handle pointer-induced aliasing. Records are accepted, but no information is given on their fields. This decision was made to simplify the design of the debugger, but records can be handled without much trouble. Jumps to local and non-local labels are fully supported. It should be emphasized that although this system is efficient, it is only a research prototype and a test-bed for new ideas.

4.3 Implementation

Syntox consists of approximately 20.000 lines of *C*, 4.000 of which implement a user-friendly interface under the *X Window* system, with its own integrated editor. Once a program has been analyzed, the user can click on any statement and the debugger pops up a window displaying run-time properties holding after the execution of the selected statement (fig. 2). If needed, the window can be dragged to a permanent position on the screen. When a procedure has reference parameters, *Syntox* gives a description of all the possible "alias sets" of this procedure [2], that is, all the subsets of variables having the same address in each procedure activation. Intermittent and invariant assertions can be inserted before every statement.

The analysis of a program is done in several steps. The first step consists in writing the *intraprocedural semantic equations* associated with each procedure of the program. The forward system of equations directly follows from the syntax of the program, and the backward equations are built by a trivial inversion of the forward system [4, 6].

The debugger then repeatedly performs a forward analysis and two backward analyses (one for each kind of assertion) and stops after a user-selectable number of passes. The default is to perform a forward analysis, two backward analyses and a final forward analysis, which is sufficient, in practice, to find most interesting bugs. However, the example given at the end of section 3 shows that this is not always the case, and it is sometimes necessary to continue the analysis one or several steps further.

Each analysis consists of a fixed point computation (either a least fixed point or a greatest fixed point) with a *widening* phase and a *narrowing* phase. Widening and narrowing [3, 4, 7, 10, 12] are standard *speed-up techniques* of abstract interpretation that can be used to compute safe approximations of fixed points when the height of

the lattice of program properties is infinite or very large, as for the interval lattice. These techniques transform possibly infinite, but exact, iterative computations of least or greatest fixed points into finite, but approximated ones. For instance, the computation of a fixed point over the lattice of intervals, which can require up to 2^n iterations when integers are coded on n bits, is reduced to at most four iterations, which makes an entire analysis only sixteen times more complex than constant propagation.

Note that when the program has recursive procedures, the interprocedural call graph is dynamically unfolded during the analysis, and each procedure activation is "duplicated" according to the value of its *token* [2, 3, 4, 19] which consists of the static calling site of the activation and the set of all its aliases. For instance, the analysis of program "Fact" would create two instances of function "F".

The duplication of procedures according to their calling sites has proven to be very useful since calling sites tend to be associated to well-defined contexts and analysing the behavior of a procedure for each context separately leads to very precise results. Of course, when this duplication is too costly in terms of time and memory, it is possible to avoid it, at the cost of a loss of precision.

4.4 Algorithms and complexity

Two fixed point computation algorithms are used by *Syntox*. Each algorithm is based on a hierarchical decomposition of the control flow-graph into strongly connected components and sub-components, described in Bourdoncle [4, 5], which defines two chaotic iteration strategies [5, 8] as well as admissible sets of widening points, that is, control points were "generalizations" take place to avoid infinite computations when working with lattices of infinite height [6, 12].

The first algorithm is used for intraprocedural analysis, for which the control flow graph is known in advance, and attempts to "stabilize" sub-components each time a component is stabilized. When the graph is acyclic, the complexity of this algorithm is linear, and when the graph is strongly connected, its worst-case complexity is the product of the height h of the abstract lattice by the sum of the individual depths of the n nodes in the decomposition of the graph, which is always bounded by:

$$\frac{h \cdot n \cdot (n+1)}{2}$$

Note that the use of widening and narrowing techniques over the lattice of intervals leads to the same complexity with $h = 4 \cdot v$, where v is the number of variables of the program. The second algorithm is used for interprocedural analysis and is based on a depth-first visit of the (dynamically unfolded) interprocedural control flow graph. Its worst-case complexity for a program with n control points, c procedure calls, p procedures and l intraprocedural loops is at most:

$$h \cdot n \cdot (c+p+l) \quad = \quad \rho \cdot h \cdot n^2$$

where $\rho \leq 1$ is the sum $l/n + c/n$ of the densities of intraprocedural loops and procedure calls in the program, and of the inverse of the average size n/p of procedures. However, practice shows that this quadratic bound is rarely reached, except for programs which

```
program BinarySearch;
    type index = 1..100;
    var n : index;  key : integer;
        T : array [index] of integer;
    function Find(key : integer) : boolean;
        var m, left, right : integer;
    begin
        left := 1;
        right := n;
        repeat
        ①   m := (left + right) div 2;
            if (key < T[m]) then
                right := m − 1
            else
                left := m + 1
        until (key = T[m]) or (left > right);
        Find := (key = T[m])
    end;
begin
    read(n, key);
    writeln("Found = ", Find(key))
end.
```

```
program QuickSort;
    var n : integer;
        T : array [1..100] of integer;
    procedure QSort(l, r : integer);
        var i, m, v, x : integer;
    begin
        if l < r then begin
            v := T[l]; m := l;
            for i := l + 1 to r do
                if T[i] < v then
                begin
                ②   m := m + 1; x := T[m];
                    T[m] := T[i]; T[i] := x
                end;
            x := T[m]; T[m] := T[l]; T[l] := x;
            QSort(l, m − 1);
            QSort(m + 1, r)
        end
    end;
begin
    readln(n); ① QSort(1, n);
end.
```

Figure 3: Examples

consist of tightly coupled mutually recursive procedures. The average complexity of interprocedural analyses of real-life imperative programs should thus be almost linear, since large programs are generally linear programs with local loops and relatively small groups of mutually recursive procedures.

4.5 Results

Apart from abstract debugging, another interesting use of *Syntox* is to prove that array accesses in a program are statically correct, so that a compiler need not generate the code to check that array indices are correct at run-time.

Classical methods used to perform compile-time array bound checking are always based on forward data-flow analyses [16, 17]. Indeed, it is not be obvious that backward propagation is interesting in this context. However, it is easy to see that every reference $T[i]$ to the i-th element of an array T of n elements in a program is an implicit invariant assertion $i \in [1, n]$. Therefore, when the program is incorrect, the back-propagation of these assertions gathers the correctness conditions on a few program points and gives better results everywhere else.

For instance, *Syntox* automatically shows that every array access is statically correct in an implementation of HeapSort, and that most accesses (i.e., all but one or two) are also correct in other implementations of various sorting algorithms. In particular, *Syntox* shows that if the conditions $n \leq 100$ at point ① and $m \leq 99$ at point ② are satisfied at run-time, then every array access in program "QuickSort" of figure 3 is guaranteed to

Program	Size	Memory	Time
BinarySearch	17	44 kB	0.5 sec
Fact	24	44 kB	0.5 sec
Select	61	64 kB	0.9 sec
Ackermann	72	99 kB	1.9 sec
QuickSort	92	98 kB	2.1 sec
HeapSort	96	108 kB	2.4 sec

Figure 4: Statistics

be correct. Without back-propagation, six tests instead of two must be done at run-time. Finally, if the main call to procedure "QSort" is replaced by the erroneous call "QSort(0, n)", *Syntox* shows that a necessary condition of correctness is that $n \leq 0$ at point ①.

Similarly, every array access is statically proven correct in program "Shuttle" of figure 1, which is taken from Markstein et al. [17], and in program "BinarySearch" of figure 3. Although this last program is fairly simple, the result is non-trivial since the test of the "repeat" loop does not make an explicit reference to the bounds 1 and 100.

It is important to remark that there is absolutely no magic behind this and the above results, and the properties $m, right, left \in [1..100]$ at point ① of program "BinarySearch" have been automatically inferred by the abstract debugger during the fixed point computation. These properties are thus the *results* of the fixed point computation, and definitely not "guessed" properties proven by a theorem prover. Also, remark that *Syntox* is not based on symbolic execution, since this method does not allow the automatic, i.e. non-interactive, determination of program invariants. The beauty of abstract interpretation is that program invariants are *synthesized* rather than simply proven [3].

The experimental comparison between the above programs compiled with and without run-time array bound checking shows a speed-up ranging from 30% to 40%.

Finally, note that the time and memory requirements for the abstract debugging of programs are reasonable. Figure 4 shows the size of differents examples (i.e. the total number of control points after having unfolded the interprocedural call graph), the allocated memory in kbytes, and the analysis time in seconds for a DEC 5000/200 Ultrix workstation. These results show that, in practice, the amount of time and memory required is almost linear in the size of the program, and therefore invalidate a common belief according to which static analysis of programs would be exponential.

5 Conclusions and future work

We have presented a new static, semantic-based approach to the debugging of programs that allows programmers to use invariant and intermittent assertions to statically and formally check the validity of a program, test its behavior along certain execution paths, and find the origin of bugs rather than their occurrences. We have shown that this method can be efficiently implemented with a worst-case quadratic complexity, and

shown that non-trivial bugs can be automatically discovered even when the lattice of abstract properties is fairly simple. Finally, we have presented the prototype abstract debugger *Syntox* which can be used to debug a subset of first-order *Pascal* programs.

The techniques we have developed can be directly applied to most "safe" imperative languages such as *Modula-2*, safe subsets of *Modula-3* or $C{+}{+}$, but they are also easily applicable to functional or logic programming languages.

Although we have not tried to debug large, real-life programs with *Syntox*, all experiments done to date indicate that the time and space complexity of abstract debugging lies somewhere between linear and quadratic, and that only intrinsically complex programs tend to be complex to analyze. We are therefore confident that this technique can be effectively applied to reasonably sized programs, but only experiments on the abstract debugging of real-life programs and languages will demonstrate the effectiveness of the approach.

Future work will be to implement abstract debugging in a real-world programming environment and give the programmer the ability to determine different categories of "standard" program properties (e.g. *nil* pointers, parity of integer variables, congruence relations [15], intervals, linear inequalities between variables [9], etc).

References

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman: "Compilers — Principles, Techniques and Tools", Addison-Wesley Publishing Company (1986)

[2] François Bourdoncle: "Interprocedural Abstract Interpretation of Block Structured Languages with Nested Procedures, Aliasing and Recursivity", *Proc. of the International Workshop PLILP'90*, Lecture Notes in Computer Science 456, Springer-Verlag (1990)

[3] François Bourdoncle: "Abstract Interpretation By Dynamic Partitioning", *Journal of Functional Programming*, Vol. 2, No. 4 (1992)

[4] François Bourdoncle: "Sémantiques des langages impératifs d'ordre supérieur et interprétation abstraite", *Ph. D. dissertation*, Ecole Polytechnique (1992)

[5] François Bourdoncle: "Efficient Chaotic Iteration Strategies with Widenings", *Proc. of the International Conf. on Formal Methods in Programming and their Applications*, Lecture Notes in Computer Science, Springer-Verlag (1993) *to appear*

[6] François Bourdoncle: "Abstract Debugging of Higher-Order Imperative Languages", *Proc. of SIGPLAN '93 Conference on Programming Language Design and Implementation* (1993)

[7] Patrick and Radhia Cousot: "Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", *Proc. of the 4th ACM Symp. on POPL* (1977) 238–252

[8] Patrick Cousot: "Asynchronous iterative methods for solving a fixpoint system of monotone equations", Research Report IMAG-RR-88, Université Scientifique et Médicale de Grenoble (1977)

[9] Patrick Cousot and Nicolas Halbwachs: "Automatic discovery of linear constraints among variables of a program", *Proc. of the 5th ACM Symp. on POPL* (1978) 84–97

[10] Patrick Cousot: "Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis. Analyse sémantique de programmes", *Ph. D. dissertation*, Université Scientifique et Médicale de Grenoble (1978)

[11] Patrick and Radhia Cousot: "Static determination of dynamic properties of recursive procedures", *Formal Description of Programming Concepts*, North Holland Publishing Company (1978) 237–277

[12] Patrick Cousot: "Semantic foundations of program analysis", in Muchnick and Jones Eds., *Program Flow Analysis, Theory and Applications*, Prentice-Hall (1981) 303–343

[13] Alain Deutsch: "On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications", *Proc. of the 17th ACM Symp. on POPL* (1990)

[14] Alain Deutsch: "A Storeless Model of Aliasing and its Abstractions using Finite Representations of Right-Regular Equivalence Relations", *Proc. of the IEEE'92 International Conference on Computer Languages*, IEEE Press (1992)

[15] Philippe Granger: "Static analysis of arithmetical congruences", *International Journal of Computer Mathematics* (1989) 165–190

[16] Rajiv Gupta: "A Fresh Look at Optimizing Array Bound Checking", *Proc. of SIGPLAN '90 Conf. on Programming Language Design and Implementation* (1990) 272–282

[17] Victoria Markstein, John Cocke and Peter Markstein: "Optimization of Range Checking", *Proc. of the SIGPLAN'82 Symp. on Compiler Construction* (1982) 114–119

[18] ISO/IEC 7185: "Information technology — Programming languages — Pascal", Revised 1983, Second edition (1990)

[19] Micha Sharir and Amir Pnueli: "Two Approaches to Interprocedural Data Flow Analysis" in Muchnick and Jones Eds., *Program Flow Analysis, Theory and Applications*, Prentice-Hall (1981) 189–233

[20] R.E. Tarjan: "Depth-first search and linear graph algorithms", *SIAM J. Comput.*, 1 (1972) 146–160

[21] Andrew P. Tolmach and Andrew W. Appel: "Debugging Standard ML Without Reverse Engineering", *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, ACM Press (1990) 1–12

[22] Niklaus Wirth and Kathleen Jensen: "Pascal user manual and report" (Second Ed.), Springer-Verlag (1978)